Receiver Simulink Study

Receiver Simulink Study

Objective

Building on your knowledge of the transmitter system, in this lab you will study the operation of the receiver subsystem using Simulink. Combined with your transmitter, this will form a complete communications chain. This lab will also teach you why proper synchronization is a critical component of a practical communications system. You will become acquainted with the following operations in the receiver:

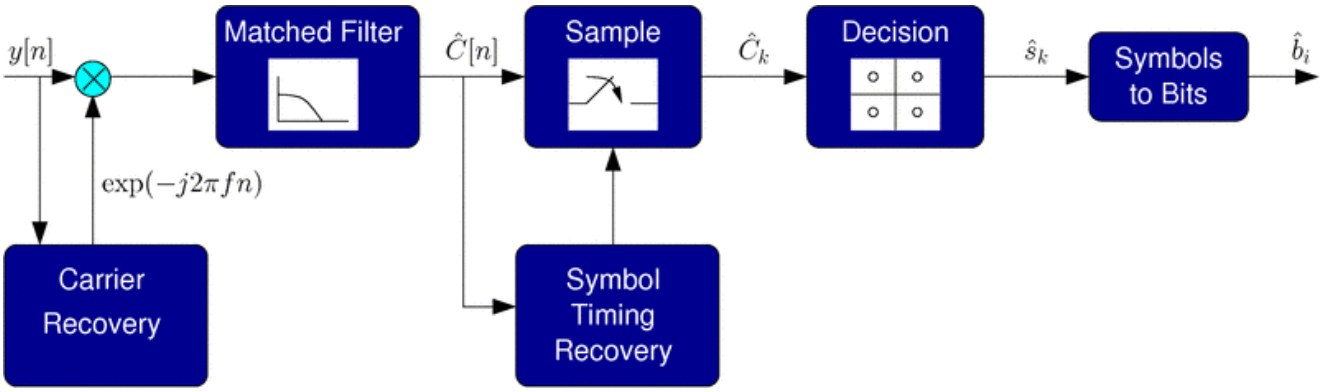
- Down conversion
- Matched filtering
- Decision rules
- Carrier and symbol timing recovery (synchronization)

Pre-lab

Again, for the pre-lab, just read through this lab outline so that you have an idea of what you will be doing. Especially make sure you understand the background section.

Background

In the last lab, we created a transmitter that took binary information, converted this to symbols, generated pulses that were weighted by the complex baseband symbol weights, and up-converted this to a carrier frequency. The receiver must now "undo" these operations, or downconvert back to baseband, sample at the appropriate times, and make a decision which symbols were transmitted. The figure below shows a diagram of a generic receiver chain:



For simplicity, the chain has been drawn using complex signals, and realize that for real processing, we would just have two branches present for I and Q processing. The basic operation of the receiver chain is

1. Multiply by the conjugate of the carrier to shift the signal back to baseband (zero center frequency).
2. Apply a (matched) filter to remove as much (additive white Gaussian) noise as possible and the image at double the carrier frequency.
3. Sample at the optimal points to obtain estimates of I/Q symbol weights.
4. Based on these samples, make a decision which symbol was transmitted.
5. Convert symbols back to bits.

Each of these blocks is described in more detail below:

Multiply Block

In the transmitter the signal was shifted up to a convenient transmit frequency suitable for the physical channel. In the receiver, we need to shift this back to baseband (zero center frequency), which is accomplished by multiplying by the conjugate of the carrier used for up-conversion. For a real input signal, we have identical images at frequency f and -f. The multiply operation shifts the one at f to 0, but shifts the other image to -2f, which is removed by subsequent filtering.

Matched Filter

In a simple communications link, we could just use any low pass filter with a suitable passband to remove the doubled frequency component from the multiply operation, yielding an estimate of the baseband signal. In real systems where noise is present, this filter should be designed to provide optimal signal-to-noise ratio. It can be shown mathematically that the optimal operation (in the important case of additive Gaussian noise) is to use a filter that is identical to the transmit pulse-shaping filter, hence the name “matched filter.”

We will see later in the lab that we can get a raised cosine response having the desirable zero-ISI property by using a root raised cosine filter in the transmit and receive. In this way, the cascaded response is a raised cosine filter and both transmit and receive have identical (matched) filters.

Sampler

The job of the sample block is to sample the receive waveform at the optimal points to get an estimate of the original I/Q symbol weights that were generated in the transmitter. In our simulation it is easy to compute the delay required for optimal sampling. In a real communications system, the sampler block will need to be told when to sample by the synchronization blocks.

Decision

Due to noise, the I/Q symbol weight estimates will not be exactly what we set in the transmit system. The decision block needs to decide what symbol was most likely transmitted based on the received I/Q sample. For lower-order modulation (like BPSK), this is usually just comparing the value to a threshold (like 0).

Symbols to Bits

This operation is just the reverse of what we did in the transmitter. We take a single symbol and map this to multiple bits.

Carrier Recovery

Unless we run a long cable from the transmitter to the receiver having the carrier frequency, or use some kind of ultra-stable atomic clock, the two carriers at transmit and receive will not be identical (and even then …). In wireless RF systems, the offset can typically be 10s to 100s of kHz, which will create significant error in our receiver chain.

To remove the carrier offset, we can recover the carrier by removing the modulation from y[n] and locking an oscillator to this signal. Alternatively, we can put a "pilot tone" in the transmit signal that is an unmodulated carrier whose phase and frequency are directly related to the modulated transmit signal. By isolating this carrier, we can easily generate the carrier for the modulated signal as well.

Carrier recovery usually uses a phase-locked loop (PLL) which can be thought of as a tracking algorithm for sinusoidal signals.

Symbol Timing Recovery

In simulation, we can compute how much delay is in our system and sample at exactly the right times to recover the transmitted symbols. In a real system, the symbol clocks will not be exactly the same at transmit and receive. Also, we do not know how much relative delay there is between transmit and receive.

To obtain the optimal sample points, we can exploit excess bandwidth of the transmit signal to "see" the optimal sample points in the I/Q waveforms. This can be done automatically by locking a PLL to the rising and falling edges of these waveforms. In the lab, you will gain some intuition how this works.

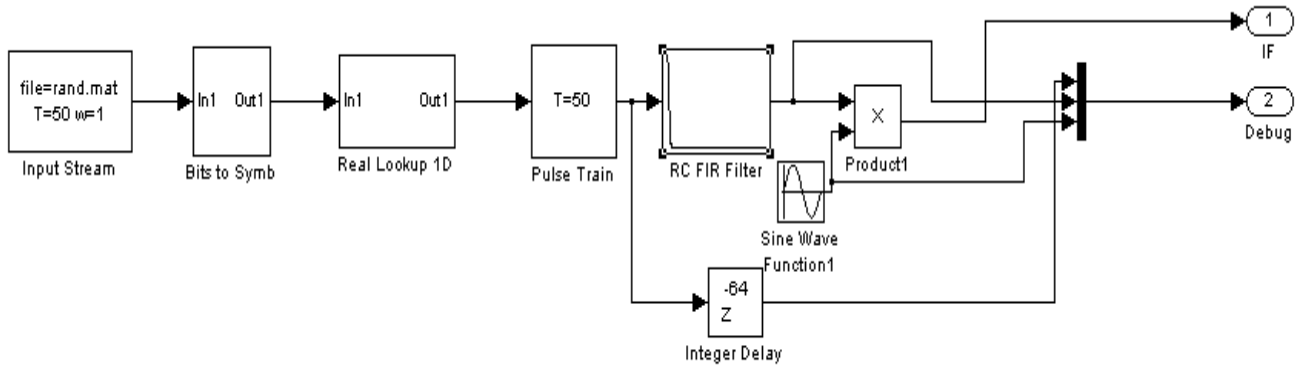
Procedure

To complete this lab, you again need access to a computer with MATLAB and Simulink. Make sure you have your transmit designs from last time, since we will use this in this lab as a starting point. You should probably create a new directory for lab 2 to help you manage things, too.

Creating a Subsystem

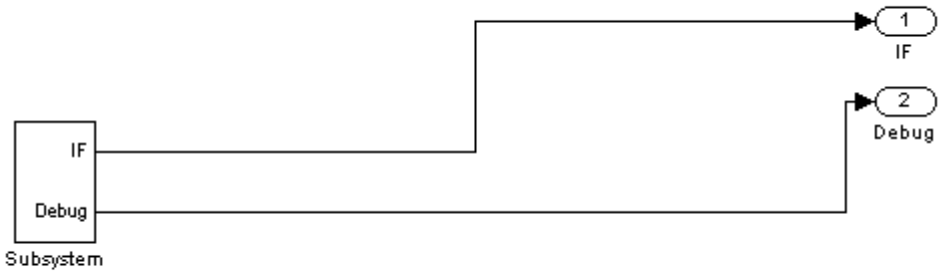
Maybe you noticed that once you have added very many blocks to your Simulink model, that things start to look pretty ugly (complicated). We can better manage complicated designs by using "subsystems." In this lab, we will put the whole transmitter inside a single subsystem block to allow us to concentrate on the receiver.

Get your final BPSK design for the transmitter from the last lab and save this under the name `bpsk_rx1.mdl`. Take the scope out of the design, and instead put the output ports (found under Ports & Subsystems) as shown below:



You should also give useful names to these output ports. The actual transmit signal is called "IF" for "intermediate frequency". This is the signal that your receiver will need to decode, and which models what would actually be received in a real communications system. The other "bundle" of signals coming out of the MUX is just called "debug", and contains the pulse train, the baseband signal, and the carrier, which will be useful to check correct operation of the decoding at the receiver.

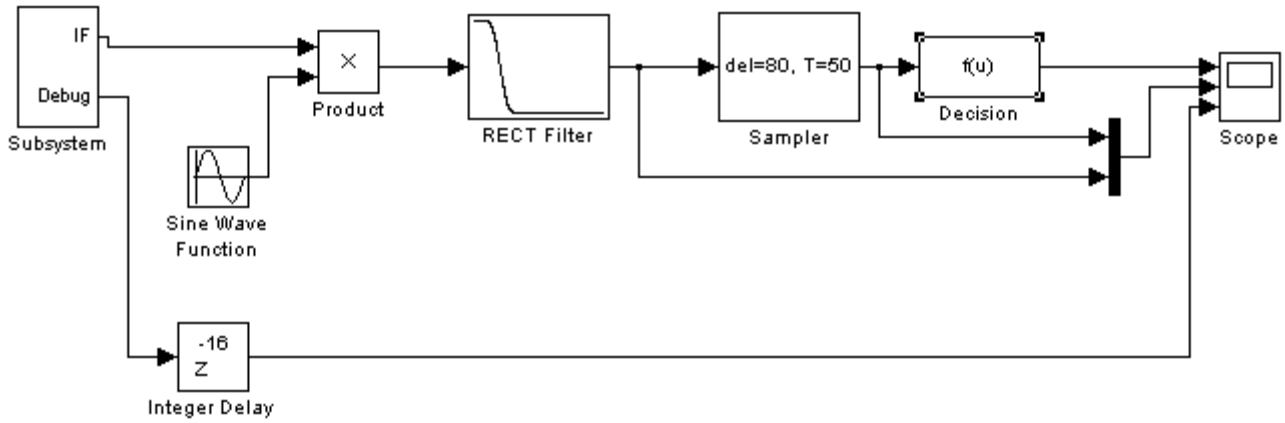
To create a subsystem, select all of the transmit blocks (draw a box around them), right click, and then select "Create Subsystem." You should now see just one block with the output ports you specified:



Now you can delete the ports that have been propagated to this higher level design and connect this to the receiver that you will design next. Note that you can modify things inside your transmit subsystem by simply double clicking on that block.

Basic BPSK Receiver

Next, create a basic BPSK receiver as depicted below.



Make sure you understand what this chain is doing by comparing it to the receiver chain explained in the background section!

Notes:

- The `sin()` block should be the same frequency as in your transmitter. I used an amplitude of 2 for this block to undo the factor of 1/2 caused by the downconversion.
- The RECT filter is a new block from the dsp_blocks library used here as a low-pass filter to remove the double frequency component. Since the bandwidth of your transmit signal is around 1/sample_rate = 1/50 = 0.02, you just need to make sure the upper cut-off frequency is above this. For parameters, I used f0=0 (low-pass), f1=0.1, and taps=32.
- The low-pass filter introduces an additional 16 samples of delay, so take this into account!
- The decision block is implemented with the `Fcn` block taken from the `User defined functions` library.

The decision block needs to take a baseband sample, and decide which symbol this is closest to. For BPSK, we can just check if the signal is positive or negative, and assign this to the bits (or symbols) 1 and 0, respectively. For example, we might do something like

```
if (input > 0),

  output = 1;

else

  output = 0;

end
```

In MATLAB, a TRUE expression evaluates to 1 and a FALSE expression evaluates to 0. So, we can do the decision operation with a compact expression like:

```
(u(1)>0)*1 + (u(1)<=0)*0
```

Here, u(1) is the input sample. If it is greater than a threshold of 0, the output is 1. If it is less than (or equal to) zero, we set the output to zero. Note that the second term is actually not needed.

Once your BPSK receiver is configured and you understand basically how it is supposed to work, try running it for 1000 samples and see if it gives what you think it should. Make sure you understand all the signals you are seeing!

Matched Filtering

Remember that we will have optimal noise performance if the receive filter is the same as your transmit filter. Try modifying your BPSK design and use the same raised cosine filter as in your transmitter. Remember, you will probably have to change the delay of the signals for sampling / plotting since this filter gives a 64 sample delay.

After making the change, rerun your design. Do you still have the zero ISI property?

Root Raised-Cosine Matched Filter

The usual way to maintain the raised cosine response (and the zero-ISI property) is to use a *root* raised cosine filter at the transmit and receive. "Root" means that we take the square root of the RC response in the *frequency domain*. This way if we cascade two root RC filters, we get the RC shape. One problem with the root raised cosine (RRC) filter is that the time response is much longer than the RC filter. So, to support the RRC filter you need to do the following:

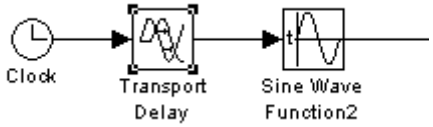
1. Change the transmit and receive filters to root filters (check the box in the parameters)
2. Make the filters 512 taps long to support the longer time response
3. Since each filter now creates 256 samples of delay, make changes to any delay blocks that depend on this.

I would like to point out that in a real system, we would not create such a long filter (512 taps) for this simple RRC operation. The reason we have to make it so long is because we are using a fixed (high) sample rate throughout our whole system. A real system would typically use the high sample rate only for the upconversion/downconversion parts and then use perhaps 1/10 of the rate for the baseband processing. In that case, the filter would be more like 64 samples long, which is more reasonable.

After making the changes, rerun your system. You should see something very similar to what you had with just one RC filter on the transmit and no matched filter at the receiver. Also, check to make sure the outputs of the sample block look correct.

Carrier Recovery

In a practical communications system, the receiver oscillator will not be the same as the transmit one. To see the effect of carriers that are not the same. Modify your transmitter so that the `sin()` block is driven with a modified clock as depicted below:

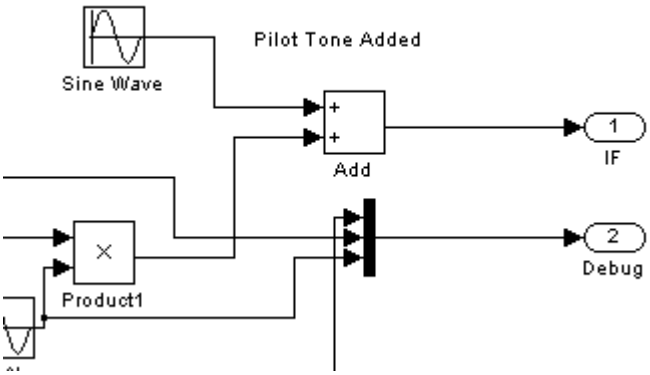


You can change the `sin()` block to use an external time reference by simply changing a box in its parameters. The clock block is found in the sources library. Also, I have found using the transport delay gives smoother plots on the oscilloscope than the discrete delay block, so that is why I used it instead.

Now, run your communications chain and see what happens to your received symbols. Can you figure out why having a delay does this? What does a delay mean in terms of the phase of the oscillators?

To make it easy for the receiver to recover the carrier, we will add a pilot tone to the transmit signal, which is simply an unmodulated component that in this case has half the frequency of our carrier. The receiver can then isolate this tone and double the frequency to get the carrier back. Any delay or frequency shift in the transmit carrier will be completely reflected in the pilot tone, allowing the receiver to track these differences.

Modify your transmit design as depicted below.

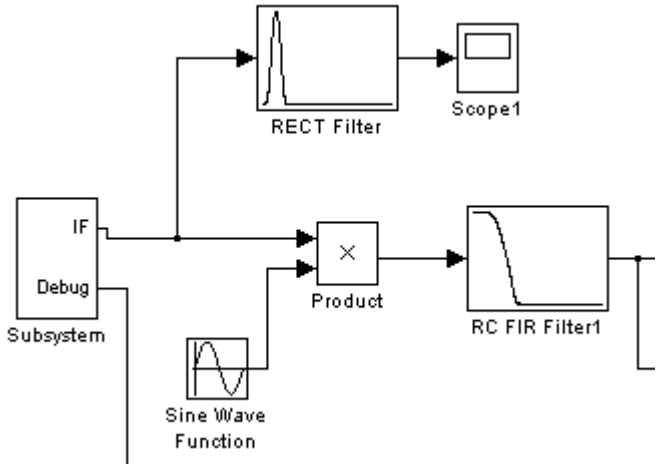


The sine wave (pilot tone) that is added to the output signal should have

- frequency = 2*pi*0.05 (discrete freq=0.05, which is half the frequency of the modulated carrier)
- amplitude = 0.05
- phase = 0

The amplitude is somewhat arbitrary, but usually we do not want to make it too high in a real system, since it wastes available transmit power. Since it is very narrowband, we can make it have fairly low amplitude and use a tight filter to remove noise.

In the receiver, we need to first isolate the pilot tone using a narrow bandpass filter. For this, you can again use the RECT filter as depicted below:



For the filter, I have used 80 taps and f0=0.03 and f1=0.07. I chose 80 taps so that the delay of the filter (40 samples) is an integer multiple of the carrier period (10 samples), making it so we don't have to adjust the phase output of the PLL.

Now put a scope on the output of the RECT filter as shown and run your chain. Do you see your carrier at f=0.05? If you can't really tell, you could also use the PSD block from before and check this way, since this shows frequency directly.

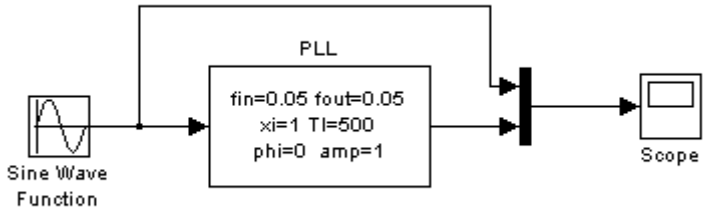
PLL-based Synchronization

Now, how do we use this carrier signal in our receiver? One idea would be to amplify the signal and use it directly instead of the sin(). This is not usually done, for a number of reasons:

1. The amplitude of the pilot signal changes as the channel between transmit and receive changes.
2. For an I/Q system, we need to also derive a signal that is 90 degrees out of phase from this.
3. Here the pilot is *half* the frequency of what we need. How do we double this? Squaring and filtering?!

A much more flexible way to generate our carrier from this signal is to use a phase-locked loop (PLL). This is a device that takes a sine wave as input and tracks its phase using a loop. The PLL generates its own sine wave internally, whose phase is locked to the incoming signal. The PLL is also free to generate any other clock or sine wave signals that are needed based on this phase. For example, the PLL can generate a carrier with double or half the frequency, apply a constant phase shift, etc.

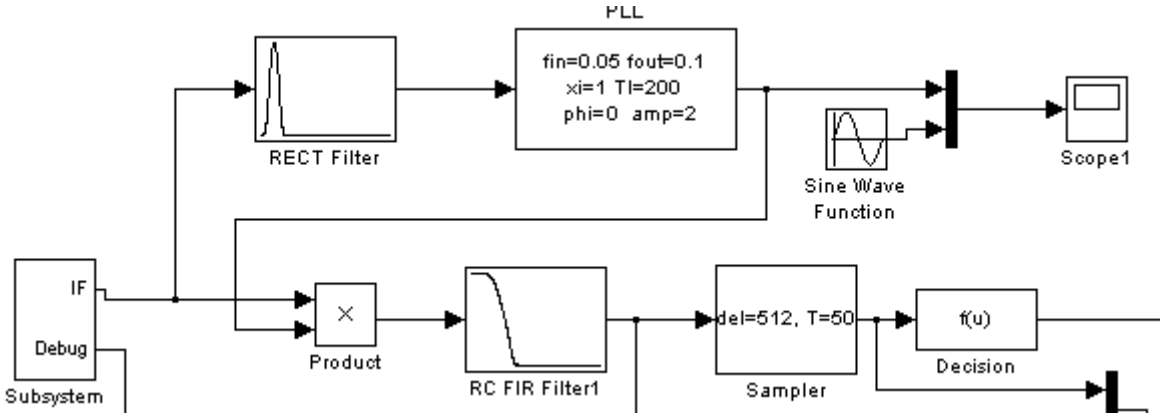
In a later lab you will learn the details of what is inside a PLL. For now, a block has already been created for you in the `DSP Blocks->Timing`menu. To understand how the PLL works, create a new empty design and put the PLL block in it with a source as shown below. Do not forget to check your model settings (Solver selection: fixed step, solver: discrete).



The key parameters are shown on the PLL block in the figure. What it means is that the PLL is expecting a nominal input frequency of 0.05 and will generate an output of the same frequency (0.05) locked to the input. The `Tl` value is the time constant of the PLL that controls how quickly the PLL should track. This should be many sinusoidal cycles to make the carrier steady (here I picked 500 samples, which means 25 periods of the carrier). Try the following things:

1. Make the source frequency 0.05 and run it. How do the input and PLL output sine wave compare?
2. Change the amplitude of the input source. Does this affect the output of the PLL?
3. Make the source frequency 0.055 or 0.45. What happens at the output of the PLL?
4. Try changing the output frequency of the PLL from 0.05 to 0.1 or 0.025. What do you see at the output? Can you see how this might be useful?

You should now be convinced that the PLL is a versatile block for synchronizing things in a communications system. We will now use it to do the carrier recovery. Place the PLL with the shown parameters in your receiver as depicted below:



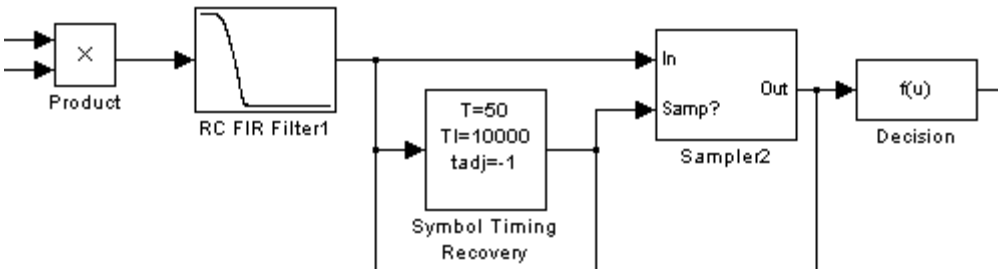
You can see that the PLL is set to double the frequency of the isolated pilot tone. The amplitude of 2 is to overcome the factor of 1/2 that occurs from the downconversion. The output of the PLL now is the carrier that we use for downconversion. The ideal carrier (the sin() block) is now just used to check if the PLL is putting out the right signal by feeding it to the scope.

Try running your receiver chain now. First look at the scope at the output of the PLL. Did the doubling operation work? Then look at the output of the receiver. Things should look like they did before. Finally, try putting a delay in your transmit subsystem or a frequency offset. Things should still work correctly!

Symbol Timing Recovery

The final thing that the communications system has to worry about is sampling the matched-filtered signal at the optimal points. As stated before, in a simulation we can just tweak these to match by considering all the delays that happen. In a real system, the clocks will be different, the transmit and receive will be powered on at different times, the channel will cause delay, etc. How can we know the delay?!

We will learn more details about symbol timing recovery in a later lab. For now, I have created a block to hide most of the details from you, which can be found in the `DSP blocks->Timing` menu. Place it in your design as shown below, and note that the lines going down are just for debugging (they go to an oscilloscope):



If you are curious, you can see what is inside by right-clicking on the block and selecting *look under mask*. It basically works by multiplying the baseband signal with a copy of itself shifted by half of the symbol time. This creates a sinusoid that has a frequency equal to the symbol rate where the zero-crossings of the sinusoid happen at precisely the optimal sample points. Cool, huh?! The PLL is then used to generate a sine wave locked to this carrier. The final delay block allows the timing to be slightly adjusted as needed. Finally, we just check for a positive zero crossing of the sine wave, which is where we should sample. This block creates a train of pulses at the optimal sample points.

The other block that is new here is the sampler block with an external enable that can be found in the `DSP Blocks`. We need this instead of the periodic sampler, since we now need to be able to have arbitrary sample points.

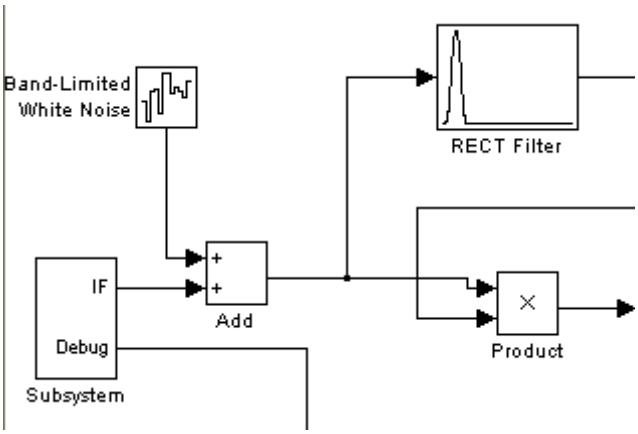
You might have noticed that the time constant is *much* longer for the PLL used for symbol timing recovery (here I picked 10000 samples). Can you think why this is needed? Remember we needed many sinusoidal cycles. What do you think we need to consider here?

Run your design with the symbol timing recovery. You will find it helpful to add the output of the symbol timing recovery block (the sample pulse train) to your scope to help you see if it is working. First run it for around 10k samples. Is it tracking yet? Now try running for a much larger value like 200k. Inspect the waveform near the end. Does it look right?

Noise

We can't leave our high-level overview of communication systems without considering noise. Noise is unwanted changes in our signal due to external sources, such as other in-band transmitters, or internal sources like thermal amplifier noise. The noise must be sufficiently small compared to the signal in order to confidently decode the transmitted signal. Noise level is usually quantified indirectly using the signal-to-noise ratio (SNR), where values like 10dB or better are normally required.

Add a small amount of noise using the bandlimited noise block (Simulink Sources menu), as shown:



Make sure you set the sample time to 1, and I have picked a value of 0.001 for the noise variance. This gives an SNR out of the matched filter around 15dB or so. Rerun your design and look at the decoding of the symbols. What do you notice that is different from before? Are the symbols still decoded properly (once the symbol timing is synchronized that is)?

Try making the noise higher (like 0.01). What happens?

Check-off

Get the TA and show the following things:

1. Show your BPSK receiver with ideal synchronization (ideal separate oscillators at transmit and receive). Explain how it works. Run it and show that the symbols (bits) are decoded correctly.
2. Show the TA your PLL testing model and show that you can double or half the frequency with it. Also show and explain the effect of changing the input frequency slightly.
3. Explain how the pilot tone is used to do carrier recovery. Show how you modified your design to support the pilot tone. Run your design with carrier recovery and show it works.
4. Explain why symbol timing recovery is needed and how your design was modified to support this. Run your design with the symbol timing recovery and show it works.
5. Answer any additional questions the TA asks.

Lab Write Up

For the write-up, provide the following things:

1. Explain briefly how the receiver works. Also, please highlight anything that was new that you didn't know before!
2. Explain how we can get a RC response by using root filters at the transmit and receive. Explain why this is preferable to using just a single (non root) RC filter somewhere.
3. Explain the need for synchronization in communications systems, and how this was accomplished in this lab. Also describe what a PLL is and what it can do. What are carrier and symbol timing recovery for?
4. Give plots showing the output of your communications system for two cases: (i) with perfect ideal synchronization, and (ii) with the synchronization blocks. Circle and label points in the printout to "prove" to the TA that it is working.
5. Describe any difficulties you experienced in getting your design to work and how you fixed these problems.